Quantum® LeaPs
innovating embedded systems

ALLIANCE PARTNER
RENESAS

QP™ Development Kit (QDK)
Renesas RX with HEW

Document Revision F
October 2012

Copyright © Quantum Leaps, LLC

info@quantum-leaps.com
www.state-machine.com

Table of Contents

1 Introduction 1
1.1 About QP™ 2
1.2 About QM™ 3
1.3 About this QDK-RX 3
1.4 Licensing QP™ 4
1.5 Licensing QM™ 5

2 Getting Started 6
2.1 Installing the QDK-RX-HEW 6
2.2 Setting up HEW Placeholders 8
2.3 Building the QP™ Libraries 8
2.4 Building the Examples 10
2.5 Running the Examples 11

3 The Renesas RX CPU 13
3.1 RX CPU Register Set 13
3.2 RX CPU Modes 14
3.3 RX CPU Stacks 14
3.4 RX CPU Interrupt Processing (Hardware) 14
3.5 RX CPU Interrupt Processing (Software) 16

4 Non-Preemptive “Vanilla” Port 17
4.1 The qep_port.h Header File 17
4.2 The qf_port.h Header File 17
4.3 The Board Support Package for the “Vanilla” Port 19

5 Preemptive QK Port 23
5.1 The qep_port.h Header File 23
5.2 The qf_port.h Header File 23
5.3 The qk_port.h Header File 23
5.4 The Board Support Package for the QK Port 25

6 QS Software Tracing Instrumentation 28
6.1 QS initialization in QS_onStartup() 28
6.2 QS Trace Output in QF_onIdle()/QK_onIdle() 29
6.3 QS Time Stamp Callback QS_onGetTime() 30
6.4 Running the QSpy host application 30

7 Related Documents and References 31

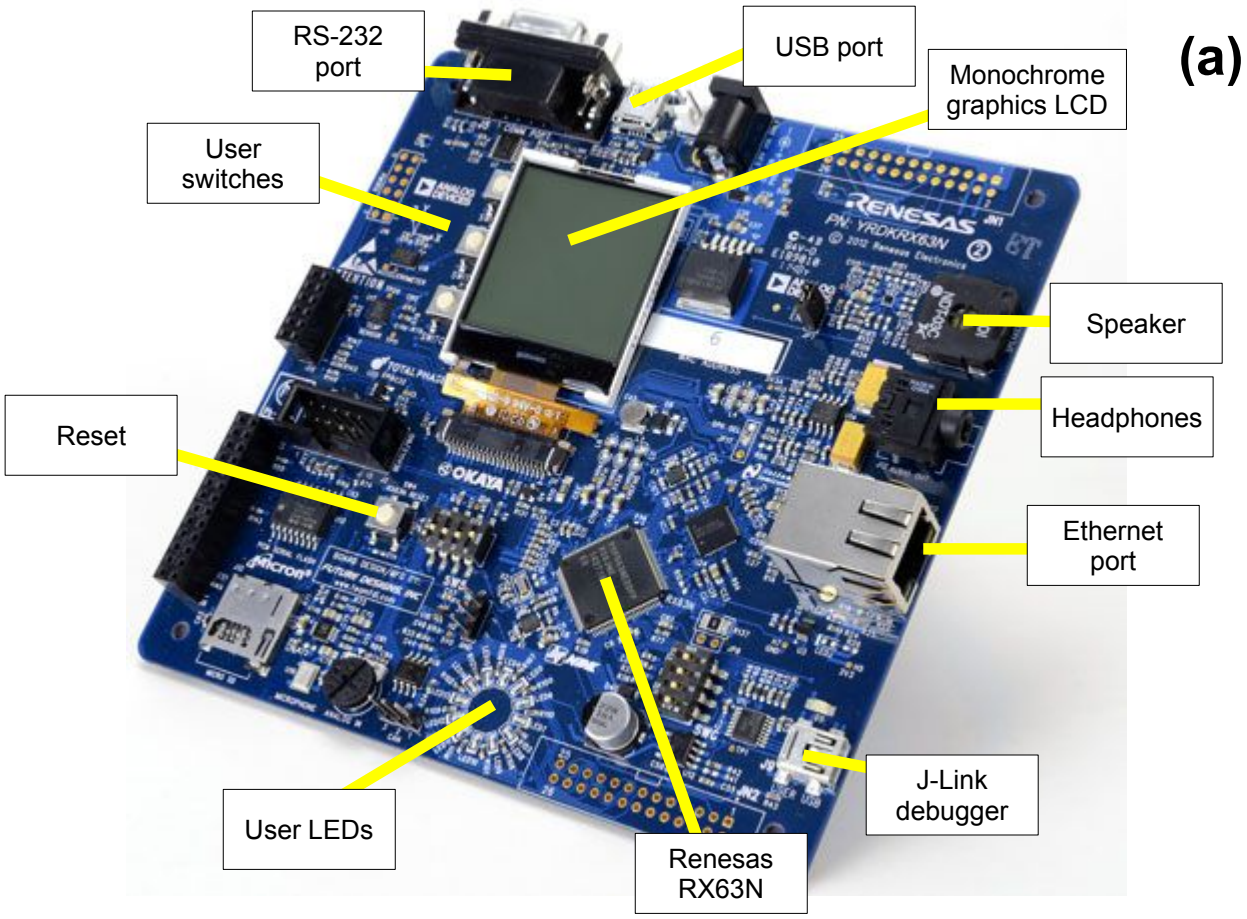
8 Contact Information 32

Copyright © Quantum Leaps, LLC. All Rights Reserved.

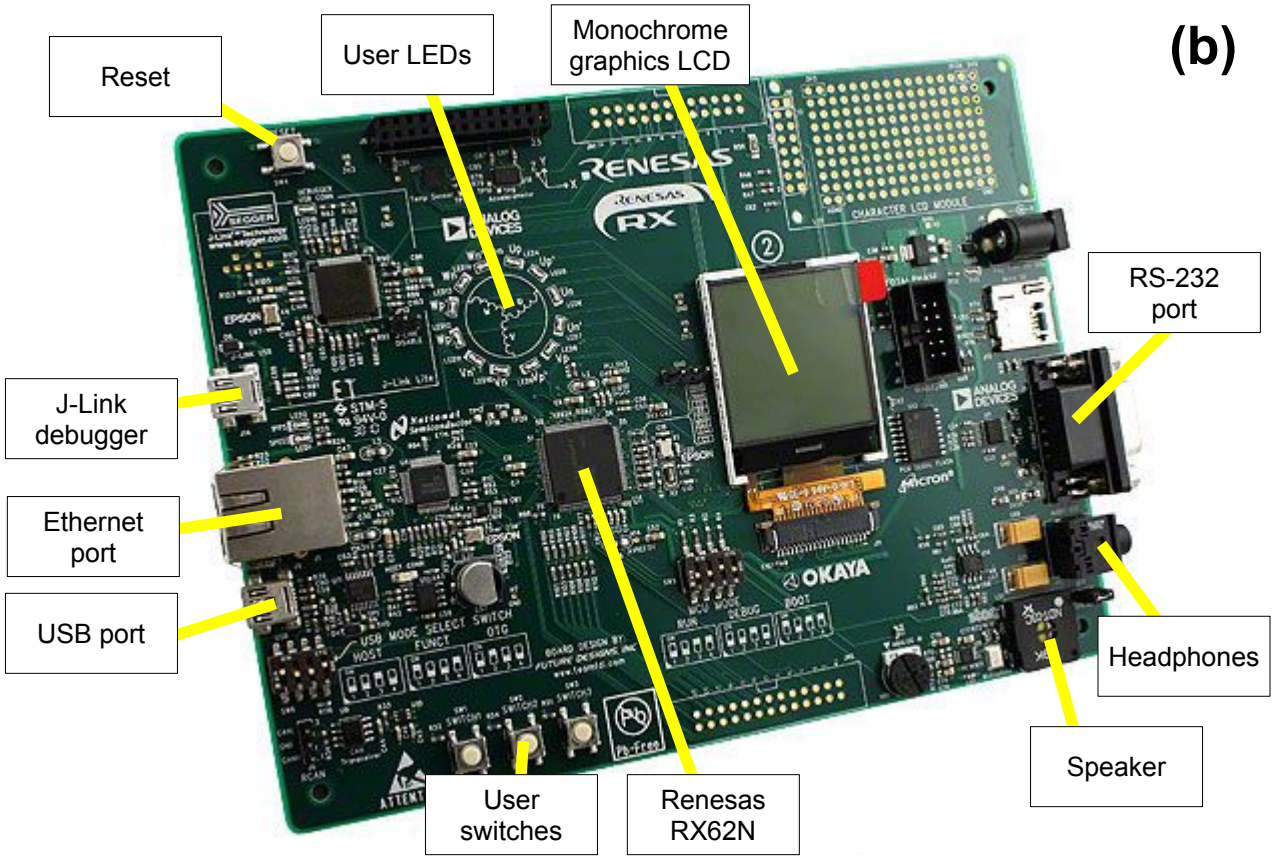
1 Introduction

This QP Development Kit™ (QDK) describes how to use the QP/C™ and QP™/C++ state machine frameworks and the QM™ modeling tool for projects based on the Renesas RX family of 32-bit processors with the Renesas High-performance Embedded Workshop (HEW). This QDK applies to the YRDKRX63N and YRDKRX62N Development Kits from Renesas shown in Figure 1(a) and Figure 1(b), respectively.

Figure 1: Renesas YRDKRX63N (a) and YRDKRX62N (b) Development Kits



Copyright © Quantum Leaps, LLC. All Rights Reserved.



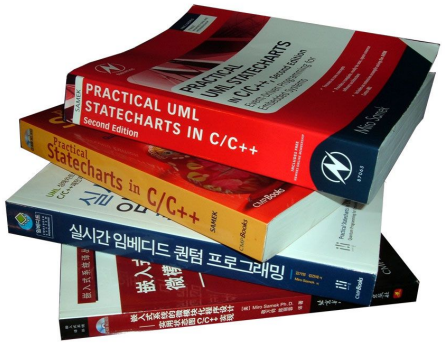
The actual hardware/software used to test this QDK is described below:

- Renesas YRDKRX63N or YRDKRX62N Development Kits (Figure 1(a) and Figure 1(b), respectively).
- Renesas HEW (High-performance Embedded Workshop) **4.x**
- Renesas RX C/C++ Compiler **1.2.1** or GNU-RX (developed by KPIT Cummings) **12.01**
- QP/C or QP/C++ version **4.5.02** or higher.

> **NOTE**: The following text refers to the Renesas-RX compiler, but the discussion applies equally to the GNU-RX compiler as well, because both toolsets are used via the Renesas HEW. The only difference is that the ports and examples for the Renesas-RX toolset are located in the directories `...\rx\renesas\...`, whereas the equivalent ports and examples for the GNU-RX toolset are located in the directories `...\rx\gnu\...`

1.1 About QP™

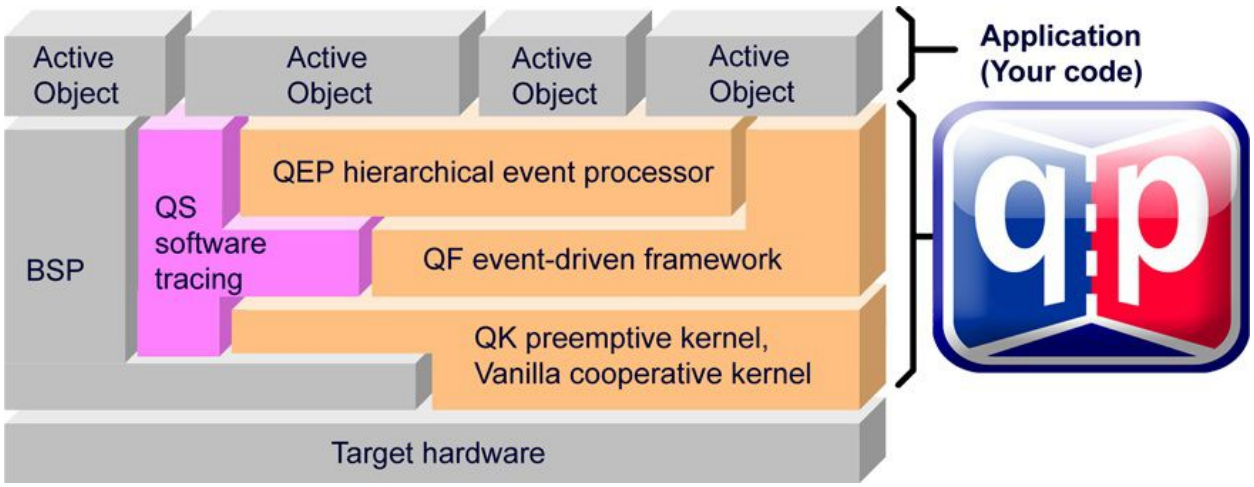
QP™ is a family of very lightweight, open source, state machine-based frameworks for developing event-driven applications. QP enables building well-structured embedded applications as a set of concurrently executing hierarchical state machines (UML statecharts) directly in C or C++, or by means of the QM™ graphical UML modeling tool. QP is described in great detail in the book "*Practical UML Statecharts in C/C++, Second Edition: Event-Driven Programming for Embedded Systems*" [PSiCC2] (Newnes, 2008).



Copyright © Quantum Leaps, LLC. All Rights Reserved.

As shown in Figure 2, QP consists of a universal UM-compliant event processor (QEP), a portable real-time framework (QF), a tiny run-to-completion kernel (QK) as well as simple cooperative kernel (Vanilla), and software tracing instrumentation (QS). Current versions of QP include: QP/C™ and QP/C++™, which require about 4KB of code and a few hundred bytes of RAM, and the ultra-lightweight QP-nano, which requires only 1-2KB of code and just several bytes of RAM. QP can manage up to 63 concurrently executing tasks structured as state machines (called active objects in UML).

Figure 2: QP Components and their relationship with the target hardware, board support package (BSP), and the application



1.2 About QM™

QM™ (QP™ Modeler) is a free, cross-platform, graphical UML modeling tool for designing and implementing real-time embedded applications based on the QP™ state machine frameworks. QM™ is available for Windows, Linux, and Mac OS X.

QM™ provides intuitive diagramming environment for creating good looking hierarchical state machine diagrams and hierarchical outline of your entire application. QM™ eliminates coding errors by automatic generation of compact C or C++ code that is 100% traceable from your design. Please visit state-machine.com/qm for more information about QM™.

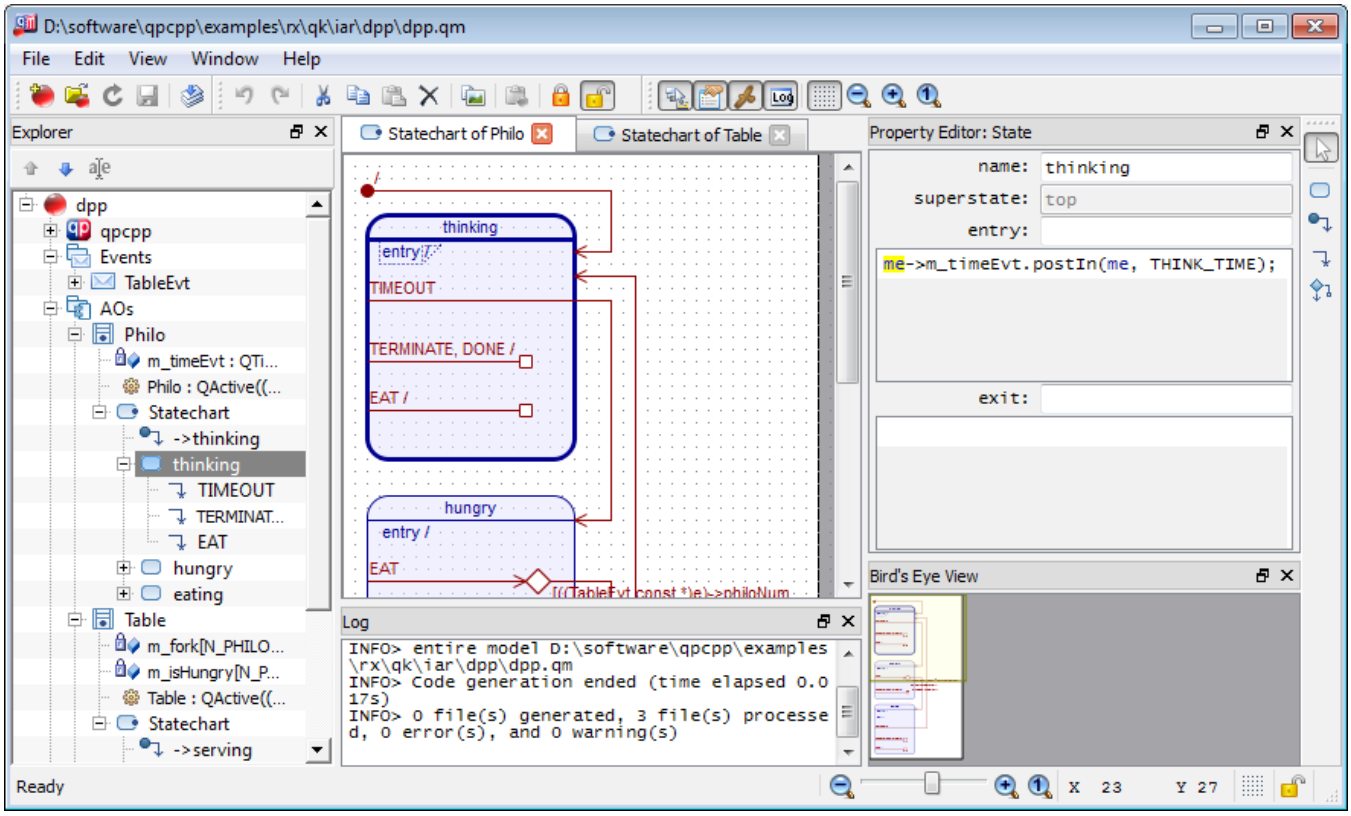
1.3 About this QDK-RX

This QDK provides working examples of code running under both the cooperative Vanilla kernel and the preemptive QK kernel. The example code is based on the Dining Philosopher Problem (DPP) sample application described in Chapter 7 of [PSiCC2] as well as in the Application Note “Dining Philosopher Problem” [QL AN-DPP 08] (included in the example code distribution).

The entire source code included with this QDK can be edited manually in a traditional code editor. However, significant parts of the code have been generated **automatically** by the QM™ modeling tool from the `dppp.qm` model file included in the QDK. The preferred way of developing QP™ applications is to make all the changes in the model and generate the code automatically.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

Figure 3: The example model opened in the QM™ modeling tool



The QDK-RX example code includes the following components:

- Board support package (BSP) which provides interrupt service routines (ISRs), all QP callbacks, and an interface to the board's LEDs, push buttons, system clock tick timer, and serial port.
- QP port to RX for the Vanilla cooperative kernel described in Chapter 7 of [PSiCC2]
- QP port to RX for the preemptive run-to-completion QK kernel described in Chapter 10 of [PSiCC2]
- The DPP example for both the cooperative Vanilla kernel and the preemptive QK kernel.
- The QM™ model of the Dinging Philosophers Problem (see Figure 3)

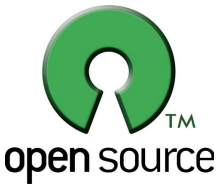
NOTE: The significant parts of the source code (files `dpp.h`, `philo.c`, and `table.c`) have been generated by the QM™ modeling tool from the `dppp.qm` model, which is the same for the Vanilla and QK versions of the DPP application. These files can be edited by hand (after unchecking the read-only property), but the changes made at the code level won't be incorporated back into the model.

NOTE: This QDK Manual covers both the C and C++ versions of the QP™ state machine frameworks. Most of the code listings in this document refer to the QP/C version. Occasionally the C code is followed by the equivalent C++ implementation to show the C++ differences whenever such differences become important.

1.4 Licensing QP™

The **Generally Available (GA)** distribution of QP™ available for download from the www.state-machine.com/downloads website is offered with the following two licensing options:

Copyright © Quantum Leaps, LLC. All Rights Reserved.



- The GNU General Public License version 2 (GPL) as published by the Free Software Foundation and appearing in the file `GPL.TXT` included in the packaging of every Quantum Leaps software distribution. The GPL *open source* license allows you to use the software at no charge under the condition that if you redistribute the original software or applications derived from it, the complete source code for your application must be also available under the conditions of the GPL (GPL Section 2[b]).

- One of several Quantum Leaps commercial licenses, which are designed for customers who wish to retain the proprietary status of their code and therefore cannot use the GNU General Public License. The customers who license Quantum Leaps software under the commercial licenses do not use the software under the GPL and therefore are not subject to any of its terms.

For more information, please visit the licensing section of our website at: www.state-machine.com/licensing

1.5 Licensing QM™

The QM™ graphical modeling tool available for download from the www.state-machine.com/downloads website is **free** to use, but is not open source. During the installation you will need to accept a basic End-User License Agreement (EULA), which legally protects Quantum Leaps from any warranty claims, prohibits removing any copyright notices from QM, selling it, and creating similar competitive products.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

2 Getting Started

This section describes how to install, build, and use the QDK-RX-Renesas. This section assumes that you have the YRDKRX62N board, and have downloaded and installed the Renesas HEW IDE and Renesas RX Toolset on your machine. Also this section assumes that you have downloaded and installed the **QP™ Baseline Code**, available for a separate download from www.state-machine.com/downloads.

> **NOTE**: To avoid repetitions, every QDK™ contains only the QP ports and example(s) pertaining to the specific MCU and compiler, but does not include the platform-independent baseline code of QP™, which is available for a separate download. It is strongly recommended that you read Chapter 12 in [PSiCC2] before you start with this QDK™. The QP directory structure is described in detail in a separate Quantum Leaps Application Note: "QP Directory Structure" [QL AN-Directory 07]).

2.1 Installing the QDK-RX-HEW

The QDK code is distributed in a ZIP archive (`qdkc_rx-renesas_yrdkrx62n.zip` for QP/C and `qdkcpp_rx-renesas_yrdkrx62n.zip` for QP/C++, for the Renesas-RX compielr, and `dkc_rx-gnu_yrdkrx62n.zip` for QP/C and `qdkcpp_rx-gnu_yrdkrx62n.zip` for QP/C++, for the GNU-RX compielr). You need to unzip this archive into the same directory, into which you've installed QP. The following Listing 1 shows the directory structure and selected files after you complete the QP and QDK-RX-Renesas installation. (Please note that the QP directory structure is described in detail in a separate Quantum Leaps Application Note: "QP Directory Structure").

Listing 1: Selected directories and files after installing QP and the QDK-RX-HEW code. The boldface indicates directories and files included in the QDK.

```
qpc\                      - QP installation directory
  +-doc\
  | +-AN_DPP.pdf            - Application Note "Dining Philosopher Problem Example"
  | +-QDK_RX-HEW.pdf        - This QDK Manual "QDK RX with HEW"
  |
  +-include\                - QP platform-independent include files
  +-qep\                    - QEP platform-independent source code
  +-qf\                     - QF platform-independent source code
  +-qk\                     - QK platform-independent source code
  +-qs\                     - QS platform-independent source code
  |
  +-examples\               - subdirectory containing the QP examples
  | +-rx\                   - Renesas RX examples
  | | +-vanilla\            - Ports to the non-preemptive "vanilla" kernel
  | | | +-renesas\          - Renesas-RX compiler (gnu for the GNU-RX compiler)
  | | | | +-dpp_yrdkrx62n\  - DPP example for (non-preemptive) for YRDKRX62N
  | | | | | +-Debug\        - directory containing the Debug build
  | | | | | | +- dpp_yrdkrx62n.abs - image of the DPP application
  | | | | | +-Release\      - directory containing the Release build
  | | | | | +-Spy\          - directory containing the Spy build
  | | | | | |
  | | | | | +-dbsct.c       - Definition of program sections
  | | | | | +-intprg.c      - Interrupt Service Routines (ISRs)
  | | | | | +-resetprg.c    - Reset exception handler (startup code)
  | | | | | +-stacksct.h    - Stack size specification
  | | | | | +-vect.h        - Interrupt vector declarations
  | | | | | +-vecttbl.c     - Fixed vector table for the RX CPU
  | | | | | |
```

Copyright © Quantum Leaps, LLC. All Rights Reserved.

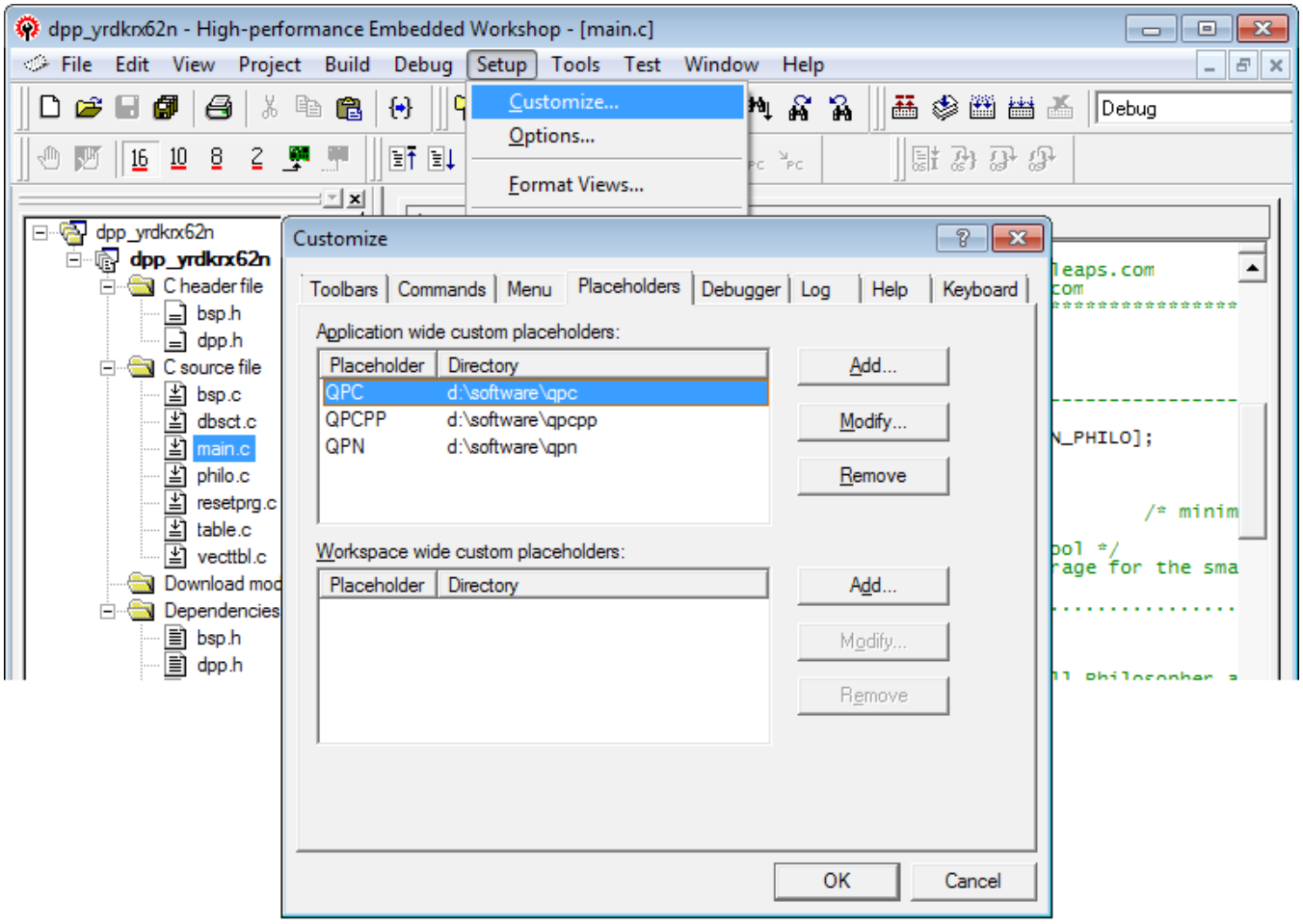
```
| | | | | +-bsp.c         - BSP for the YRDKRX62N (non-preemptive)
| | | | | +-bsp.h         - BSP header file
| | | | | +-main.c        - the main function
| | | | | +-dpp.h         - the DPP application header file (auto-generated by QM)
| | | | | +-dpp.qm        - QM model for the DPP application
| | | | | +-dpp_qk_yrdkrx62n.hwp - HEW project for the DPP application
| | | | | +-dpp_qk_yrdkrx62n.hws - HEW workspace for the DPP application
| | | | | +-philo.c       - the Philosopher active objects (auto-generated by QM)
| | | | | +-table.c       - the Table active object (auto-generated by QM)
| | | | |
| | +-qk\                 - Ports to the preemptive QK kernel
| | | +-renesas\          - Renesas-RX compiler (gnu for the GNU-RX compiler)
| | | | +-dpp-qk-yrdkrx62n\ - DPP example for (preemptive) for YRDKRX62N
| | | | | +-Debug\        - directory containing the Debug build
| | | | | | +-dpp.out     - image of the DPP application
| | | | | | +-dpp.map     - map file of the DPP application
| | | | | +-Release\      - directory containing the Release build
| | | | | +-Spy\          - directory containing the Spy build
| | | | | |
| | | | | +-bsp.c         - BSP for the YRDKRX62N (preemptive)
| | | | | +-bsp.h         - BSP header file
| | | | | +-main.c        - the main function
| | | | | +-dpp.h         - the DPP application header file (auto-generated by QM)
| | | | | +-dpp.qm        - QM model for the DPP application
| | | | | +-dpp_yrdkrx62n.hwp - HEW project for the DPP application
| | | | | +-dppk_yrdkrx62n.hws - HEW workspace for the DPP application
| | | | | +-philo.c       - the Philosopher active objects (auto-generated by QM)
| | | | | +-table.c       - the Table active object (auto-generated by QM)
| |
+-ports\                  - subdirectory containing the QP ports
| +-rx\                   - Renesas RX ports
| | +-vanilla\            - Ports to the non-preemptive "Vanilla" kernel
| | | +-renesas\          - Renesas-RX compiler (gnu for the GNU-RX compiler)
| | | | +-Debug\          - Debug build
| | | | | +-qp_rx600.lib - QP library for RX600 core (Vanilla kernel)
| | | | +-Release\        - Release build
| | | | +-Spy\            - Spy build
| | | | +-qp_rx600.hwp    - HEW project for building the QP library
| | | | +-qp.hws          - HEW workspace for building the QP library
| | | | +-qep_port.h      - QEP platform-dependent public include
| | | | +-qf_port.h       - QF platform-dependent public include
| | | | +-qp_port.h       - QP platform-dependent public include
| | | | +-qs_port.h       - QS platform-dependent public include
| | +-qk\                 - QK (preemptive kernel) ports
| | | +-renesas\          - Renesas-RX compiler (gnu for the GNU-RX compiler)
| | | | +-dbg\            - Debug build
| | | | | +-qp_rx600.lib - QP library for RX600 core (QK kernel)
| | | | +-rel\              - Release build
| | | | +-spy\            - Spy build
| | | | +-qp_rx600.hwp    - HEW project for building the QP library
| | | | +-qp.hws          - HEW workspace for building the QP library
| | | | +-qep_port.h      - QEP platform-dependent public include
| | | | +-qf_port.h       - QF platform-dependent public include
| | | | +-qk_port.h       - QK platform-dependent public include
| | | | +-qp_port.h       - QP platform-dependent public include
| | | | +-qs_port.h       - QS platform-dependent public include
```

Copyright © Quantum Leaps, LLC. All Rights Reserved.

2.2 Setting up HEW Placeholders

The HEW project files included in this QDK assume that custom HEW "**Placeholders**" are set up to point to the location of the QP frameworks in your file system. Therefore it is strongly recommended that you define the custom placeholders for QP globally for all applications. To define HEW placeholders, you launch HEW and select the menu **Setup | Customize...** menu (see Figure 4). Next, in the "Customize" dialog box, you click the "Placeholder" tab and add the placeholders for all installed QP framework types. The placeholder names needed for the provided HEW projects are as follows: **QPC** for QP/C, **QPCPP** for QP/C++, and **QPN** for QP-nano, respectively.

Figure 4: Setting "Application wide" Placeholders the HEW IDE



2.3 Building the QP™ Libraries

All QP components are deployed as libraries that you statically link to your application. The pre-built QP libraries are provided inside the `<qp>\ports\rx\` directory, so you don't need to build the QP libraries if you use the same toolset and the standard QP configuration. This section describes steps you need to take to rebuild the libraries yourself.

NOTE: The QP libraries and QP applications can be built in the following three **build configurations**:

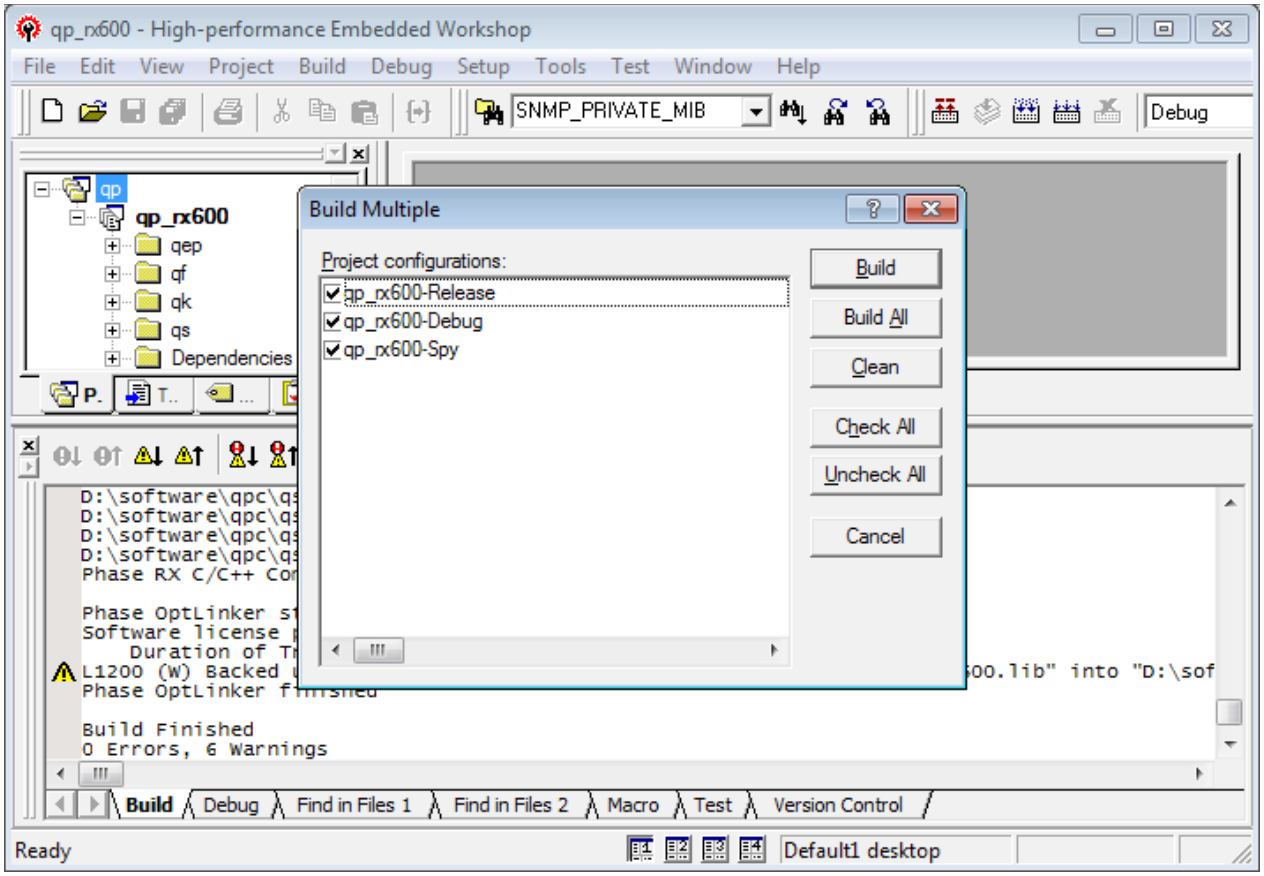
Copyright © Quantum Leaps, LLC. All Rights Reserved.

Debug - this configuration is built with full debugging information and minimal optimization. When the QP framework finds no events to process, the framework busy-idles until there are new events to process.

Release - this configuration is built with no debugging information and high optimization. Single-stepping and debugging is effectively impossible due to the lack of debugging information and optimized code, but the debugger can be used to download and start the executable. When the QP framework finds no events to process, the framework puts the CPU to sleep until there are new events to process.

Spy - like the debug variant, this variant is built with full debugging information and minimal optimization. Additionally, it is build with the QP's Q-SPY trace functionality built in. The on-board serial port and the Q-Spy host application are used for sending and viewing trace data. Like the Debug configuration, the QP framework busy-idles until there are new events to process.

Figure 5: Building the QP libraries from the HEW IDE



The DPP-RX-Renesas contains the HEW workspace qp.hws and project file qp_rx600.hwp located in the <qp>**ports**\rx\... directory. For example, to build the QP libraries for the RX600 core with the QK kernel, you launch the HEW IDE and open the workspace <qpc>**ports**\rx\qk\renesas\qp.hws. You can then batch-build all three configurations of the project qp_rx600.hwp, as shown in Figure 5.

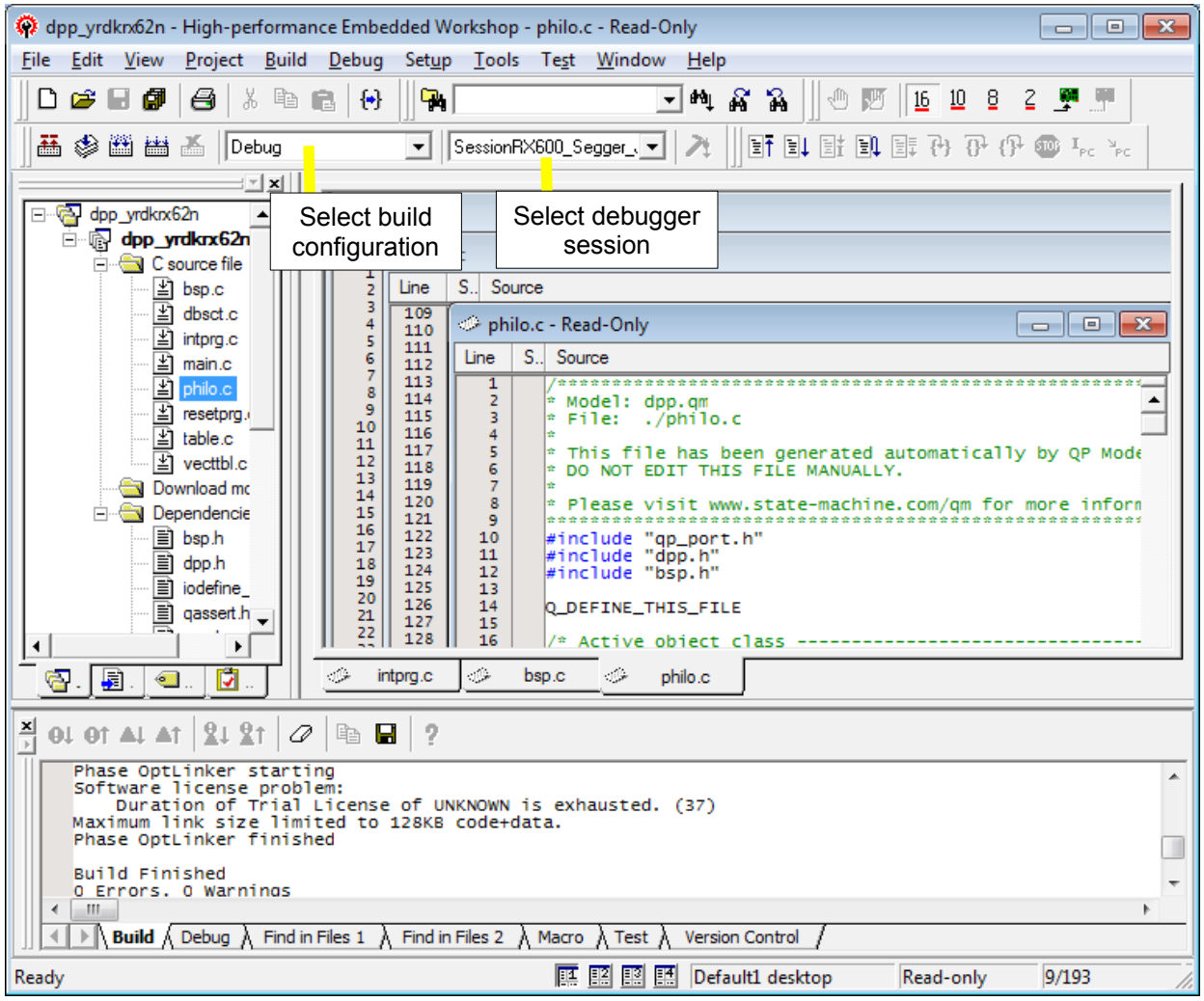
The build process should produce the QP libraries in the locations: <qp>**ports**\rx\qk\renesas\-**Debug**\qp_rx600.lib, <qp>**ports**\rx\qk\renesas**Release**\qp_rx600.lib, and <qp>**ports**\-rx\qk\renesas**Spy**\qp_rx600.lib.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

2.4 Building the Examples

The examples accompanying this QDK-RX-Renesas are based on the DPP application (see Quantum Leaps Application Note: "Dining Philosophers Problem Application" [QL AN-DPP 08] included in this QDK). The example directory `qpc\examples\rx\` contains the HEW workspaces and project that you can load into the HEW IDE, as shown in Figure 6.

Figure 6: HEW IDE with the DPP example project



For example, to build the DPP example for the cooperative Vanilla kernel perform the following steps:

- Launch HEW and open the workspace `dpp_yrdkrx62n.hws` (located in `qpc\examples\rx\vanilla\renesas\dpp-yrdkrx62n\`)
- Build the project by select Buld | Build menu or by pressing F7. The project file contains three build configurations Debug, Release, and Spy. You can select the build configuration by means of the drop-down list on the HEW toolbar, as shown in Figure 6.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

2.5 Running the Examples

You program the code into the flash memory of the MCU through the HEW Session. First, you must select the debugger session **SesstionRX600_Segger_JLink** (see Figure 6) and you need to connect to the target by selecting the Debug | Connect menu. Next, in HEW you must always make sure that the image is actually loaded into the target. You download the code into the target by selecting the Debug | Download Modules menu.

After you download the code, the HEW debugger will stop the program at `PowerON_Reset-PC()`. Please press the Go toolbar button in the HEW to let the program continue. The User LEDs of the YRDKRX63N board should start blinking (see Figure 1). The LEDs are assigned as follows: LED4, LED5, LED6, LED7, and LED8 show the status of Dining Philosophers 0-4, respectively. These LEDs are on when the corresponding Philosopher is in the "eating" state, otherwise the LEDs are off.

The User LED12 is used to visualize the idle loop activity. The brightness of the LED12 is proportional to the frequency of invcations of the idle loop. The LED12 is always toggled with interrupts disabled, so no interrupt execution time contributes to the brightness of the User LED12.

This version of the DPP application also demonstrates the user input to the system. By pressing and holding down the SWITCH1 of the YRDKRX63N board, you can **pause the philosophers**, so that they will not get permissions to eat. This means that after a while all philosophers will end in the "hungry" state (so that LED4-LED8 will turn off). After releasing the SWITCH1, the philosophers get the permissions to eat, so the application resumes as normal. The PAUSED state of the application is visualized by the LED11.

The LCD display is not used in the DPP example, even though the backlight of the LCD is on. On the YRDKRX63N board the LCD backlight is hard-wired and is not controlled by the MCU, so it cannot be turned off.

2.5.1 Q-SPY Software Tracing

QS is a software tracing facility built into all QP components and also available to the Application code. QS allows you to gain unprecedented visibility into your application by selectively logging almost all interesting events occurring within state machines, the framework, the kernel, and your application code. QS software tracing is minimally intrusive, offers precise time-stamping, sophisticated runtime filtering of events, and good data compression (see Chapter 11 in [PSiCC2]).

To see the QS software trace output, you need to connect the RS-232 Port of the YRDKRX62N board (see Figure 1) to the COM port of your PC with a straight-through RS-232 cable. Alternatively, you can use an RS-232-to-USB converter connected directly to the RS-232 Port of the YRDKRX62N board.

In the HEW IDE you need to switch to the Spy configuration and download it to the target board. Next you need to launch the QSPY host utility in the Windows console to observe the output in the human-readable format.

> **NOTE**:The QSPY host utility is part of the **Qtools** collection, available for download from http://www.state-machine.com/downloads/index.php#QTools. After installing Qtools, you should add the installation directory to the PATH environment variable.

You launch the QSPY utility on a Windows PC :

```
qspy -c COM9
```

This will start the QSPY host application to listen on COM2 serial port with the standard baud rate 115200. (Please use the actual virtual COM port number on your PC.) The screen shot in Figure 7 shows the QSPY output from the DPP run:

Copyright © Quantum Leaps, LLC. All Rights Reserved.

Figure 7: Screen shot from the QSPY run

Administrator: Command Prompt
TICK : Ctr= 25
0000046876 QK_sche: prio= 5, pin= 0
0000046876 Disp==>: Obj=l_philo[4] Sig=TIMEOUT_SIG Active=Philo::thinking
0000046878 QK_sche: prio= 6, pin= 5
0000046878 Disp==>: Obj=l_table Sig=HUNGRY_SIG Active=Table::serving
0000046879 User000: 4 hungry
0000046881 User000: 4 eating
0000046882 Intern : Obj=l_table Sig=HUNGRY_SIG Source=Table::serving
Q_ENTRY: Obj=l_philo[4] State=Philo::hungry
0000046884 ==>Tran: Obj=l_philo[4] Sig=TIMEOUT_SIG Source=Philo::thinking New=Philo::hungry
0000046885 QK_sche: prio= 5, pin= 0
0000046885 Disp==>: Obj=l_philo[4] Sig=EAT_SIG Active=Philo::hungry
Q_ENTRY: Obj=l_philo[4] State=Philo::eating
0000046887 ==>Tran: Obj=l_philo[4] Sig=EAT_SIG Source=Philo::hungry New=Philo::eating
0000046888 QK_sche: prio= 4, pin= 0
0000046889 Disp==>: Obj=l_philo[3] Sig=TIMEOUT_SIG Active=Philo::thinking
0000046890 QK_sche: prio= 6, pin= 4
0000046891 Disp==>: Obj=l_table Sig=HUNGRY_SIG Active=Table::serving
0000046892 User000: 3 hungry
0000046893 Intern : Obj=l_table Sig=HUNGRY_SIG Source=Table::serving
Q_ENTRY: Obj=l_philo[3] State=Philo::hungry
0000046894 ==>Tran: Obj=l_philo[3] Sig=TIMEOUT_SIG Source=Philo::thinking New=Philo::hungry
0000046896 QK_sche: prio= 4, pin= 0
0000046896 Disp==>: Obj=l_philo[3] Sig=EAT_SIG Active=Philo::hungry
0000046897 Ignored: Obj=l_philo[3] Sig=EAT_SIG Active=Philo::hungry
0000046898 QK_sche: prio= 3, pin= 0
0000046899 Disp==>: Obj=l_philo[2] Sig=TIMEOUT_SIG Active=Philo::thinking
0000046900 QK_sche: prio= 6, pin= 3
0000046901 Disp==>: Obj=l_table Sig=HUNGRY_SIG Active=Table::serving
0000046902 User000: 2 hungry
0000046904 User000: 2 eating
0000046904 Intern : Obj=l_table Sig=HUNGRY_SIG Source=Table::serving
0000046905 QK_sche: prio= 5, pin= 3
0000046906 Disp==>: Obj=l_philo[4] Sig=EAT_SIG Active=Philo::eating
0000046907 Intern : Obj=l_philo[4] Sig=EAT_SIG Source=Philo::eating
0000046908 QK_sche: prio= 4, pin= 3
0000046909 Disp==>: Obj=l_philo[3] Sig=EAT_SIG Active=Philo::hungry
0000046909 Ignored: Obj=l_philo[3] Sig=EAT_SIG Active=Philo::hungry
Q_ENTRY: Obj=l_philo[2] State=Philo::hungry
0000046911 ==>Tran: Obj=l_philo[2] Sig=TIMEOUT_SIG Source=Philo::thinking New=Philo::hungry
0000046912 QK_sche: prio= 3, pin= 0
0000046913 Disp==>: Obj=l_philo[2] Sig=EAT_SIG Active=Philo::hungry
0000046914 Ignored: Obj=l_philo[2] Sig=EAT_SIG Active=Philo::hungry
0000046915 QK_sche: prio= 3, pin= 0
0000046915 Disp==>: Obj=l_philo[2] Sig=EAT_SIG Active=Philo::hungry
Q_ENTRY: Obj=l_philo[2] State=Philo::eating
0000046917 ==>Tran: Obj=l_philo[2] Sig=EAT_SIG Source=Philo::hungry New=Philo::eating
0000046918 QK_sche: prio= 2, pin= 0
0000046919 Disp==>: Obj=l_philo[1] Sig=TIMEOUT_SIG Active=Philo::thinking
0000046920 QK_sche: prio= 6, pin= 2
0000046921 Disp==>: Obj=l_table Sig=HUNGRY_SIG Active=Table::serving
0000046922 User000: 1 hungry
0000046923 Intern : Obj=l_table Sig=HUNGRY_SIG Source=Table::serving
Q_ENTRY: Obj=l_philo[1] State=Philo::hungry
0000046924 ==>Tran: Obj=l_philo[1] Sig=TIMEOUT_SIG Source=Philo::thinking New=Philo::hungry

Copyright © Quantum Leaps, LLC. All Rights Reserved.

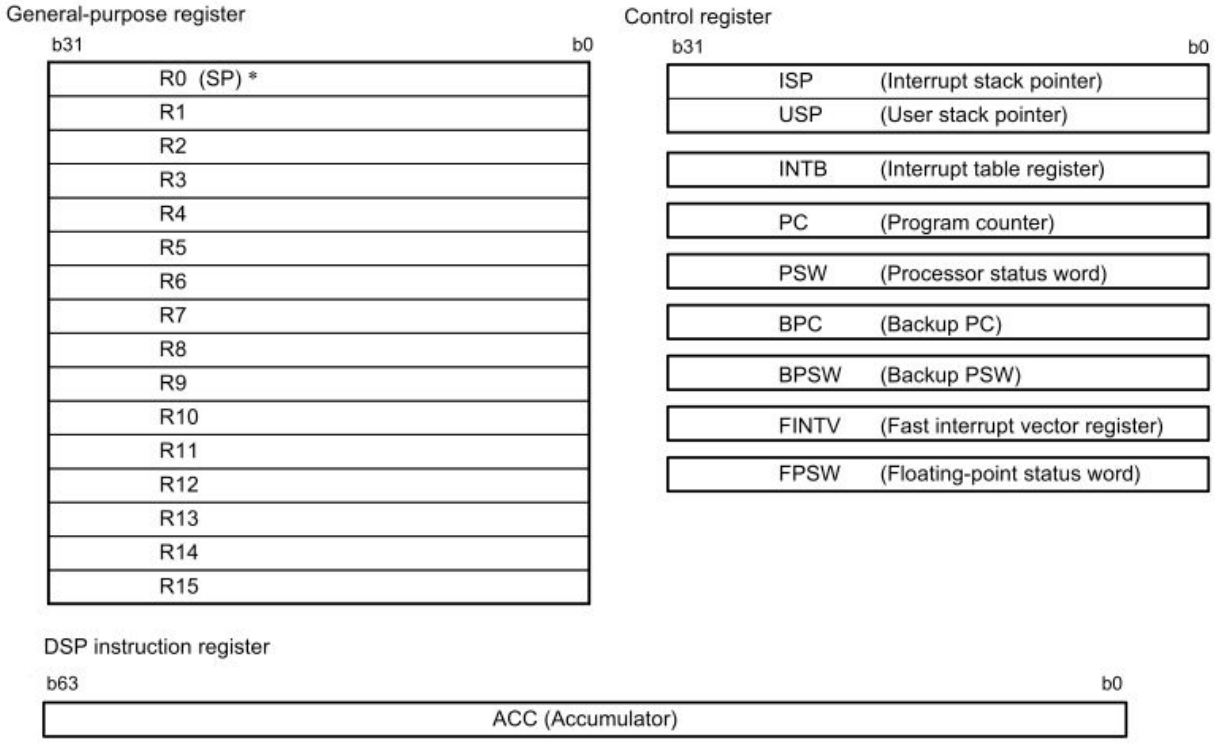
3 The Renesas RX CPU

This section provides a very quick overview of the Renesas RX CPU and its features relevant to the QP framework.

3.1 RX CPU Register Set

The RX CPU has sixteen general-purpose 32-bit registers, nine 32-bit control registers, and one 64-bit accumulator used for DSP instructions.

Figure 8: RX CPU register set



3.1.1 The Processor Status Word (PSW) Register

The 32-bit Processor Status Word (PSW) register (see Figure 9) deserves some extra attention.

Figure 9: RX Processor Status Word (PSW) register

	b31	b30	b29	b28	b27*	b26	b25	b24	b23	b22	b21	b20	b19	b18	b17	b16
	—	—	—	—	IPL[3:0]				—	—	—	PM	—	—	U	I
Value after reset:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
	—	—	—	—	—	—	—	—	—	—	—	—	O	S	Z	C
Value after reset:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Note : * Since the interrupt priority levels are from 0 to 7 for the RX610 Group, bit 27 is reserved. Writing to bit 27 is ineffective.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

The PSW has an interrupt control bit (PSW[I], bit 16) that globally enables (PSW[I] = 1) or disables (PSW[I] = 0) interrupts. Furthermore, a 4-bit field called Interrupt Priority Level (PSW[IPL], bits 24-27) reflects the current interrupt priority level. All interrupts to the CPU are assigned a priority of 0-15, with 0 being the lowest priority, and 15 being the highest priority. When PSW[I] = 1 (interrupts are enabled), any interrupt with a priority higher than the current IPL will interrupt the CPU. Interrupts of a priority less than or equal to the current IPL value are kept pending until the IPL drops.

NOTE:The RX610 series only has a 3-bit IPL field (values 0-7); bit 27 is reserved, writes are ignored. At reset, PSW[I] = 0 (interrupts disabled) and PSW[IPL] = 0 (lowest priority level).

3.2 RX CPU Modes

The RX CPU supports two processor modes: Supervisor and User. Each processor mode imposes a level on rights of access to memory and the instructions that can be executed. Supervisor mode carries greater rights than user mode and this initial mode out of reset. **All QP ports to RX CPU run exclusively in the Supervisor mode**.

3.3 RX CPU Stacks

The RX stack pointer (R0) can be either the interrupt stack pointer (ISP) or the user stack pointer (USP), depending on the value of the stack pointer select bit (U) in the processor status word (PSW). **All QP ports to RX CPU use exclusively the interrupt stack pointer (ISP)** and the user stack pointer is **not** used at all. Consequently, the user stack size should be set to zero to prevent wasting of RAM.

3.4 RX CPU Interrupt Processing (Hardware)

The RX family features a sophisticated interrupt control unit (ICU) and a prioritized hardware interrupt scheme that is a perfect fit for the QP's single-stack, run-to-completion execution architecture. Figure 10 shows the parts of the exception processing handled automatically by the hardware and the parts handled by the software.

As shown in Figure 10, the hardware processing includes the following steps executed **atomically**:

1. The exception is acknowledged/accepted by the processor
2. The processor status word (PSW) is stacked automatically
3. PSW[I] is cleared, **disabling interrupts**

NOTE: The RX processor disables interrupts automatically and atomically in hardware as part of the interrupt entry sequence.

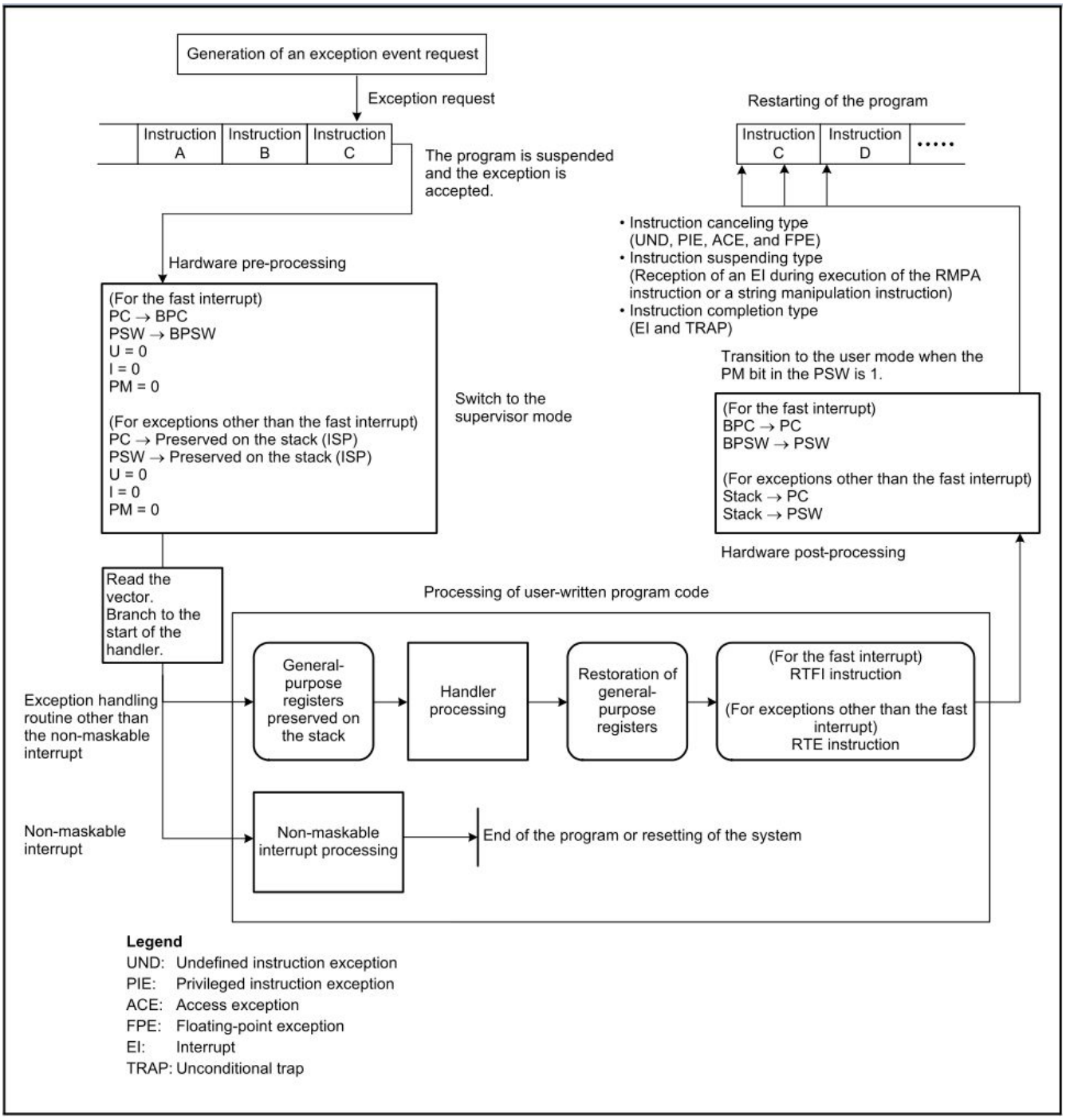
4. The program counter (PC) of the next instruction (after returning from interrupt) is stacked automatically
5. PSW[IPL] is set to the priority of the current interrupt
6. The PC is loaded with the vector from the vector table, and execution of the service routine begins

NOTE:The RX processor supports also the concept of a **fast interrupt**, which uses the backup PC (BPC) and the backup PSW (BPSW) registers instead of the stack for preserving the PC and the PSW, respectively.

The fast interrupt can be used with the cooperative “Vanilla” kernel, but should **not** be used with the preemptive QK kernel, because it does not use the machine's natural stack protocol.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

Figure 10: RX exception processing


Generation of an exception event request
Exception request
Instruction A
Instruction B
Instruction C
The program is suspended and the exception is accepted.
Hardware pre-processing
(For the fast interrupt)
PC → BPC
PSW → BPSW
U = 0
I = 0
PM = 0
(For exceptions other than the fast interrupt)
PC → Preserved on the stack (ISP)
PSW → Preserved on the stack (ISP)
U = 0
I = 0
PM = 0
Switch to the supervisor mode
Restarting of the program
Instruction C
Instruction D
• Instruction canceling type (UND, PIE, ACE, and FPE)
• Instruction suspending type (Reception of an EI during execution of the RMPA instruction or a string manipulation instruction)
• Instruction completion type (EI and TRAP)
Transition to the user mode when the PM bit in the PSW is 1.
(For the fast interrupt)
BPC → PC
BPSW → PSW
(For exceptions other than the fast interrupt)
Stack → PC
Stack → PSW
Hardware post-processing
Read the vector. Branch to the start of the handler.
Processing of user-written program code
Exception handling routine other than the non-maskable interrupt
General-purpose registers preserved on the stack
Handler processing
Restoration of general-purpose registers
(For the fast interrupt)
RTFI instruction
(For exceptions other than the fast interrupt)
RTE instruction
Non-maskable interrupt
Non-maskable interrupt processing
End of the program or resetting of the system
Legend
UND: Undefined instruction exception
PIE: Privileged instruction exception
ACE: Access exception
FPE: Floating-point exception
EI: Interrupt
TRAP: Unconditional trap

After vectoring to the Interrupt Service Routine (ISR), the software takes over (see the next section). The software returns from the ISR by executing the `RTE` instruction (or `RTFI` for the fast interrupt), at which point the hardware takes over and executes the hardware post processing. This **atomic** sequence consists of restoring the PSW and PC from the stack and resuming the interrupted program.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

3.5 RX CPU Interrupt Processing (Software)

The Renesas RX compiler, as most embedded cross-compilers, can generate interrupt service routines (ISRs), which are designated with the special `#pragma interrupt` (see Listing 2). The Renesas compiler can also automatically populate the RX vector table with the address of the ISR, when you provide the vector number by means of the `#pragma vect=...` attribute. The HEW project wizard generates the declarations of all the interrupts in the file `vect.h` header file.

Listing 2: Declaring an interrupt function with Renesas RX compiler (file `vect.h`)

```
#pragma interrupt (Excep_CMTU0_SMT0(vect=28))
void Excep_CMTU0_SMT0(void);
```

NOTE: An interrupt function must have the return type void, and it cannot specify any parameters.

You can define your own interrupt functions in the module `intprog.c`. The following Listing 3 shows the definition of the `Excep_CMTU0_SMT0()` interrupt service routine.

Listing 3: Defining an interrupt function with Renesas RX compiler (file `intprog.c`)

```
#pragma interrupt (Excep_CMTU0_SMT0(vect=28))
void Excep_CMTU0_CMT0(void) {
    QF_ISR_ENTRY();  /* inform the QF Vanilla kernel about entering the ISR */

#ifdef Q_SPY
    QS_tickTime_ += QS_tickPeriod_;       /* account for the clock rollover */
#endif
    QF_TICK(&Excep_CMTU0_CMT0);            /* process all armed time events */

    QF_ISR_EXIT();    /* inform the QF Vanilla kernel about exiting the ISR */
}
```

In case of the RX processor, such compiler-generated ISRs are perfectly suitable for the QP framework (including both the cooperative “Vanilla” kernel and the preemptive QK kernel), but in case of the QK kernel special care must be taken to preserve all the registers possibly used by the tasks. Also, for any kind of kernel, you need to understand which registers are and, more importantly, aren't saved and restored by the compiler, so that you know which operations are **not** allowed in the ISR code.

The Renesas compiler-generated `#pragma interrupt` function saves and restores any general-purpose registers clobbered by this function and returns with the `RTE` instruction. However, the interrupt function does **not** save the Accumulator or the `FPSW` registers.

NOTE: This means that the ISR cannot perform any operations that clobber the Accumulator or the `FPSW` register, such as the multiply, multiply-and-accumulate instructions; `EMUL`, `EMULU`, `FMUL`, `MUL`, and `RMPA`, or any floating-point instructions in which case the prior value in the accumulator and/or the `FPSW` register is modified by execution of the instruction.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

4 Non-Preemptive "Vanilla" Port

The "Vanilla" port shows how to use the QP frameworks on an RX-based system with the non-preemptive "Vanilla" kernel. In this version you're using the cooperative kernel built-into the QF framework and you're not using the QK component.

NOTE: The source code for the QP port to the cooperative "Vanilla" kernel is generic and should not need to change for different members of the Renesas 32-Bit MCU RX Family (RX200, RX600, etc.).

4.1 The `qep_port.h` Header File

The QEP header file for the RX port is located in `qpc\ports\rx\vanilla\renesas\qep_port.h`. Listing 4 shows the `qep_port.h header` file.

Listing 4: `qep_port.h` header file for the non-preemptive QP configuration and Renesas compiler

```
(1) #include <stdint.h>        /* exact-width integers, WG14/N843 C99, 7.18.1.1 */

    #include "qep.h"              /* QEP platform-independent public interface */
```

(1) The Renesas Compiler provides the C99-standard exact-width integer types in the standard `<stdint.h>` header file.

4.2 The `qf_port.h` Header File

The QF port header file for the RX family is located in `qpc\ports\rx\vanilla\renesas\qf_port.h`. This file specifies the configuration constants for QF (see Chapter 8 in [PsiCC2]) as well as the unconditional interrupt disabling/enabling and the critical section policy.

The most important porting decision you need to make in the `qf_port.h` header file is the critical section policy. The RX family of MCUs allows using the simplest "unconditional interrupt unlocking" policy (see Section 7.3.2 of [PSiCC2]), because RX

Listing 5: qf_port.h header file for the non-preemptive QP configuration and Renesas compiler

```
                      /* The maximum number of active objects in the application */
(1) #define QF_MAX_ACTIVE               63
                         /* The maximum number of event pools in the application */
(2) #define QF_MAX_EPOOL                6
                                         /* QF interrupt disabling/enabling */
(3) #define QF_INT_DISABLE()            clr_psw_i()
(4) #define QF_INT_ENABLE()             set_psw_i()

                                               /* QF critical section entry/exit */
(5) /* QF_CRIT_STAT_TYPE not defined: unconditional interrupt unlocking, NOTE01 */
(6) #define QF_CRIT_ENTRY(dummy)        QF_INT_DISABLE()
(7) #define QF_CRIT_EXIT(dummy)         QF_INT_ENABLE()

                                           /* QF ISR entry/exit, also see NOTE01 */
(8) #define QF_ISR_ENTRY()              QF_INT_ENABLE()
(9) #define QF_ISR_EXIT()               QF_INT_DISABLE()
```

Copyright © Quantum Leaps, LLC. All Rights Reserved.

```
                                                              /* inline definitions */
(10) #pragma inline_asm clr_psw_i
(11) static void clr_psw_i(void) { CLRPSW I }

(12) #pragma inline_asm set_psw_i
(13) static void set_psw_i(void) { SETPSW I }

     #include "qep_port.h"                                        /* QEP port */
     #include "qvanilla.h"                        /* "Vanilla" cooperative kernel */
     #include "qf.h"                  /* QF platform-independent public interface */
```

(1) The `QF_MAX_ACTIVE` macro specifies the maximum number of active object priorities in the application. You always need to provide this constant. Here, `QF_MAX_ACTIVE` is set to the maximum limit of 63 active object priorities in the system. You can decrease this number to reduce the RAM footprint of the QF framework.

(2) The `QF_MAX_EPOOL` macro specifies the maximum number of event pools in the application.

(3) The `QF_INT_DISABLE()` macro resolves to the function `clr_psw_i()`, which is defined in inline assembly as `"CLRPSW I"`. This single instruction clears PSW[I] in the PSW register, thereby disabling all interrupts.

(4) The `QF_INT_ENABLE()` macro resolves to the function `set_psw_i()`, which is defined in inline assembly as `"SETPSW I"`. This single instruction sets PSW[I] in the PSW register, thereby enabling all interrupts.

(5) The `QF_CRIT_STAT_TYPE` macro is not defined, which means that the simple critical section policy of "unconditional interrupt locking and unlocking" is applied.

(6) The `QF_CRIT_ENTRY()` macro does not use its parameter in this case and simply unconditionally disables interrupts.

(7) The `QF_CRIT_EXIT()` macro does not use its parameter in this case and simply unconditionally re-enables interrupts.

NOTE: The simple and very efficient critical section policy does **not** allow nesting of critical sections. However, the RX interrupt handlers disable interrupts upon entry, so the body of the ISR becomes a critical section. To avoid nesting of critical sections, interrupts must be enabled before calling any QP service from an ISR. This enabling interrupts after entry to the ISR and disabling interrupts before exit is accomplished in the macros `QF_ISR_ENTRY()`/`QF_ISR_EXIT()` described below.

---

(8) To avoid nesting of critical sections, the macro `QF_ISR_ENTRY()` must be called at the beginning of every ISR before invoking any QP services. This macro re-enables interrupts (i.e., permit interrupt nesting and preemption of the current ISR by higher-priority interrupts). Please note, however that the prioritized interrupt controller of the RX CPU prevents any interrupts of lower or equal priority from preempting the currently serviced interrupt level.

NOTE: If you don't wish interrupts to be able to preempt each other, you can always assign the same priorities to all interrupts.

---

(9) The macro `QF_ISR_EXIT()` must be used at the very end of every ISR. This clears the CPU's PSW[I] bit to 0, just as it was when the ISR was entered.

(10-11) The Renesas RX compiler requires this unusual syntax to define the inline assembly function `clr_psw_i()`.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

(12-13) The Renesas RX compiler requires this unusual syntax to define the inline assembly function set_psw_i().

4.3 The Board Support Package for the "Vanilla" Port

The Board Support Package (BSP) for a QP application implements the board initialization, ISRs, QP callback functions, and application-specific board-specific functions. The whole BSP is located in the file bsp.c in the directory qpc\examples\rx\vanilla\renesas\dpp-yrdkrx62n\.

4.3.1 Interrupt Service Routines (ISRs) for the "Vanilla" kernel

As mentioned earlier, the Renesas compiler enables the user to write ISRs as interrupt functions in C/C++. These compiler-generated ISRs are perfectly adequate for the non-preemptive Vanilla kernel.

Listing 6: Tick timer ISR for the Vanilla kernel in intprog.c

```
(1) #include "vect.h"
(2) #pragma section IntPRG

    . . .

(3) void Excep_CMTU0_CMT0(void) {
(4)     QF_ISR_ENTRY();  /* inform the QF Vanilla kernel about entering the ISR */

(5) #ifdef Q_SPY
(6)     l_tickTime_ += l_tickPeriod_;         /* account for the clock rollover */
    #endif

(7)     QF_TICK(&l_tickISR);                   /* process all armed time events */

(8)     QF_ISR_EXIT();    /* inform the QF vanilla kernel about exiting the ISR */
    }
```

(1) The header file vect.h provides declarations of all possible interrupt functions.

(2) All ISRs must be placed in the IntPRG section.

(3) The ISR must be one of the #pragma interrupt functions declared in the vect.h header file.

(4) The macro QF_ISR_ENTRY() leaves the critical section established automatically by the RX hardware upon the entry to the interrupt. This allows calling the QP services (such as QF_TICK()) outside the critical section.

(5) This code is conditionally compiled into the build when the macro Q_SPY is defined. When using QS the variable l_tickTime_ keeps track of how many have elapsed so far. At the time an actual time-stamp is needed, the free running counter is added to this value to get a high resolution 32-bit time-stamp.

(6) The macro QF_TICK() is used to notify the framework about another clock tick. This allows the framework to perform time-event management activities. This call may result in one or more events being posted to active objects which had started timers. The parameter to QF_TICK() is only used when tracing with QSPY, otherwise it is ignored.

(7) The macro QF_ISR_EXIT() re-establishes again the critical section by restoring PSW[I] to the value it was upon first entering the ISR.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

4.3.2 Saving and Restoring the Accumulator in the ISRs

The interrupt functions generated by the RX compiler with `#pragma vector` do **not** preserve the RX accumulator register, unless the compiler option `-save_acc` is activated. This can be achieved in the HEW IDE by selecting the menu **Build | RX Standard Toolchain**, the **CPU** tab, **Details...** button (see Figure 11).

> **NOTE**: When using the Vanilla kernel, you need to save and restore the accumulator register if the ISR body can clobber it somehow, for example by performing one of the MAC instructions or a floating-point instruction.

4.3.3 Specifying the Stacks (stacksct.h header file)

As described before, the QP framework uses exclusively the Supervisor stack and does not use the User Stack at all. In the Renesas compiler the sizes of the stacks and heap are specified by means of `#pragma stacksize` directives, which traditionally are placed in the `stacksct.h` header file. The `#pragma stacksize si` specifies the Supervisor.

Listing 7: Specifying stack size int stacksct.h

```
#pragma stacksize si=256
```

4.3.4 Idle loop customization in the Vanilla port

As described in Chapter 7 of [PSiCC2], when no events are available, the non-preemptive Vanilla scheduler invokes the platform-specific callback function `QF_onIdle()`, which you can use to save CPU power, or perform any other idle processing (such as Q-SPY software trace output).

> **NOTE**: The idle callback `QF_onIdle()` must be invoked with interrupts disabled, because the idle condition can be changed by any interrupt that posts events to event queues. `QF_onIdle()` must internally unlock interrupts, ideally atomically with putting the CPU to the power-saving mode (see also Chapter 7 in [PSiCC2]).

Listing 8: Idle callback QF_onIdle() for the Vanilla kernel in bsp.c

```
(1) void QF::onIdle(void) {                    // entered with interrupts DISABLED

(2)     BSP_LED_IDLE_ON();      // toggle the User LED on and then off, see NOTE02
(3)     BSP_LED_IDLE_OFF();

(4) #ifdef Q_SPY
        . . .
(5) #elif defined NDEBUG
(6)     __wait_for_interrupt();
(7) #else
(8)     QF_INT_ENABLE();
    #endif
    }
```

(1) This routine is always entered with interrupts disabled to avoid a race condition with interrupts that could post one or more events to active objects and thus invalidate the idle condition.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

(2-3) The idle LED (LED12 of the YRDKRX62N board) is toggled on and off each time through the idle callback. Thus, a brighter LED indicates more idle time. A heavily loaded CPU will spend little or no time in the idle loop and thus the idle LED would be dim. Note that in a release build, the CPU will sleep most of the time, and the LED will be toggled only at the rate of the interrupts that wake up the CPU. In a debug build variant, the CPU never sleeps, and the LED's intensity gives a good indication of the idle task activity.

(4) The software-tracing code is conditionally compiled into the build when enabled at compile time. When using QS, the trace data is sent out during idle loop processing. If the UART's Transmit Data Register (TDR) is empty, this code calls a QSPY API to see if there is another byte of data to send out the serial port.

(5) In release mode (no debugger) the CPU can be put to low-power sleep.

(6) The `__wait_for_interrupt()` intrinsic provided by the Renesas compiler simply inserts a `WAIT` instruction to the RX CPU. (More advanced power management could be performed here; this port only sleeps the CPU, and does not put any other peripheral in low-power mode).

NOTE: Even though the WAIT instruction is executed with interrupts locked (PSW[I] = 0), the instruction re-enables interrupts (PSW[I] = 1) as the CPU goes to sleep. Thus, when the CPU re-wakes, interrupts are already enabled.

(7) release build variant) or doing QSPY tracing, simply unlock interrupts (a requirement) before returning.

4.3.5 *Assertion-Failure Callback*

As described in Chapter 6 of [PSiCC2], all QP components use internally assertions to detect errors in the way application is using the QP services. You need to define how the application reacts in case of assertion failure by providing the callback function `Q_onAssert()`. Typically, you would put the system in a fail-safe state and try to reset. It is also a good idea to log some information as to where the assertion failed.

The following listing shows the `Q_onAssert()` callback in the `bsp.c` file. The function locks all interrupts and hangs in an endless loop. When executing the code from a debugger you can break into the code and inspect the cause of the assertion by backtracking the call stack.

NOTE: This policy is only adequate for testing, but is **not** adequate for production release.

Listing 9: Assertion-Failure callback in bsp.c

```
void Q_onAssert(char const Q_ROM * const Q_ROM_VAR file, int line) {
    (void)file;                                   /* avoid compiler warning */
    (void)line;                                   /* avoid compiler warning */
    QF_INT_DISABLE();
    for (;;) {
    }
}
```

4.3.6 *The Startup Code (resetprg.c)*

The startup code is traditionally provided in the file `resetprg.c`, which provides the definition of the `PowerON_ResetPC()` reset vector. The standard starup code is adequate for QP, except that you should remove switching the RX CPU mode to User, because the QP port is designed to execute in the Supervisor mode.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

Listing 10: Startup code in resetprg.c

```
#pragma section ResetPRG
#pragma entry PowerON_Reset_PC

void PowerON_Reset_PC(void) {
    set_intb((unsigned long)__sectop("C$VECT"));
    set_fpsw(0x00000100);

    _INITSCT();

//    _INIT_IOLIB(); // Remove the comment to use SIM I/O
//    errno=0;       // Remove the comment when you use errno
//    srand(123);    // Remove the comment when you use rand()
//    _s1ptr=NULL;   // Remove the comment when you use strtok()

      _CALL_INIT();  // Remove the comment when you use global class objects
      main();

//    _CLOSEALL();   // Remove the comment to use SIM I/O
//    _CALL_END();   // Remove the comment when you finalize class objects

      brk();
}
```

Copyright © Quantum Leaps, LLC. All Rights Reserved.

5 Preemptive QK Port

This section describes how to use QP with the preemptive QK real-time kernel described in Chapter 10 of [PSiCC2]. The benefit is very fast, fully deterministic task-level response and that execution timing of the high-priority tasks (active objects) will be virtually insensitive to any changes in the lower-priority tasks. The downside is bigger RAM requirement for the stack. Additionally, as with any preemptive kernel, you must be very careful to avoid any sharing of resources among concurrently executing active objects, or if you do need to share resources, you need to protect them with the QK priority-ceiling mutex (again see Chapter 10 of [PSiCC2]).

NOTE: The preemptive configuration with QK uses more stack than the non-preemptive Vanilla configuration. You may need to adjust the Supervisor mode stack size to be large enough for your application.

As mentioned earlier, the RX family's interrupt control and handling is a perfect match for a single-stack run-to-completion kernel. As in the Vanilla kernel, also under the QK kernel the CPU executes in Supervisor Mode using the Interrupt Stack Pointer. User Mode, and the User Stack Pointer, are unused.

As with the Vanilla port, the QK allows nested interrupts by (re-)enabling interrupts inside ISRs once it is safe. It is the application developer's responsibility to assign interrupt priorities correctly in order to achieve the desired behavior. Interrupts of the same level or lower cannot preempt each other. By setting all interrupts to the same priority (e.g., 1), interrupt nesting is effectively disabled.

5.1 The `qep_port.h` Header File

The QEP header file for the QK port is located in `qpc\ports\rx\qk\renesas\qep_port.h`. It is identical to the non-preemptive calse shown in Listing 4.

5.2 The `qf_port.h` Header File

The QEP header file for the QK port is located in `qpc\ports\rx\qk\renesas\qf_port.h`. It is almost identical to the non-preemptive calse shown in Listing 5, except the functions for saving and restoring the RX Accumulator are moved to the QK port `qk_port.h`.

5.3 The `qk_port.h` Header File

As with any preemptive kernel, the QK needs to be notified about entering an interrupt context and about exiting an interrupt context in order to perform a context switch, if necessary. Additionally, in case of the RX CPU and the specific interrupt functions generated by the Renesas compiler, the QK kernel must perform an **extended context switch** to preserve the state of the FPSW (Floating Point Status Word) register. The coprocessor support via the extended context switch of the QK kernel is described in Section 10.4 of [PsiCC2].

Listing 11: ISR entry and exit in qk_port.h

```
                                        /* QK interrupt entry and exit */
(1) #define QK_ISR_ENTRY() do { \
(2)     RxAcc acc_; \
(3)     save_acc(&acc); \
(4)     ++QK_intNest_; \
(5)     set_psw_i()

(6) #define QK_ISR_EXIT() \
(7)     clr_psw_i(); \
```

Copyright © Quantum Leaps, LLC. All Rights Reserved.

```
 (8)      --QK_intNest_; \
 (9)      if (QK_intNest_ == (uint8_t)0) { \
(10)          uint8_t p = QK_schedPrio_(); \
(11)          if (p != (uint8_t)0) { \
(12)              set_psw((unsigned long)0); \
(13)              QK_schedExt_(p); \
              } \
          } \
(14)      restore_acc(&acc_); \
      } while (0)

(15)  #define QK_EXT_SAVE(act_) \
(16)      ((act_)->thread = (void *)get_fpsw())

(17)  #define QK_EXT_RESTORE(act_) \
(18)      set_fpsw((unsigned long)(act_)->thread)

      #include <machine.h>      /* for intrinsics set_ipl()/get_fpsw()/set_fpsw() */
      #include "qk.h"                 /* QK platform-independent public interface */
```

(1) The `QK_ISR_ENTRY()` macro must be invoked at the entry to the ISR code and definitely before any QP services.

(2) The storage for the RX Accumulator is allocated on the stack.

(3) The RX Accumulator is saved.

(4) With interrupts still disabled, the QK interrupt nesting level is incremented to account for entering another level of interrupt. (The QK interrupt nesting level is decremented and checked on ISR exit, discussed later.)

(5) Interrupts are enabled via a compiler intrinsic function (remember that RX CPU enters the ISR with interrupts disabled). Note that this only sets PSW[I] = 1 (enabled); it does not adjust the PSW[IPL], the processor's interrupt priority level. Only interrupts with a priority greater than the current IPL value may preempt this ISR.

(6) The `QK_ISR_EXIT()` macro and must be the last line of a user-written ISR in C. The macro performs "exit ISR housekeeping" and transfers control to the QK scheduler if necessary.

(7) Interrupts are disabled (PSW[I] set to 0) in order to safely adjust the interrupt nesting level.

(8) The QK interrupt nesting level is decrement to account for leaving the interrupt. When the interrupt nesting value becomes 0, this indicates that the code is about to return to task level.

(9-11) The `QK_schedPrio_()` function returns the highest priority task ready to run or zero if no task has a higher priority than the current level.

(12) The Interrupt Priority Level in the Program Status Word register (PSW[IPL]) is set to 0, which corresponds to the **task level** priority. This is exactly what is described in [PSiCC2], Figure 10.2, item (6); setting IPL to zero corresponds to executing an "End of Interrupt" instruction, which means that the interrupt controller of the RX CPU stops prioritizing the current interrupt. At this point, the interrupt controller will allow any interrupt to run, which is exactly the behavior expected at task level.

(13) The extended QK scheduler is invoked to keep launching the high-priority tasks as long as they are above the currently serviced priority (see [PSiCC2], Figure 10.2, items (6) through (8)). The QK scheduler is designed to be called with with interrupts disabled and also returns with interrupts disabled, although it enables interrupts before launching any task.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

NOTE: The **extended** QK scheduler is used, instead of the regular scheduler, to save and restore the `FPSW` register. The following macros `QK_EXT_SAVE()`/`QK_EXT_RESTORE()` specify how to save and restore the extended context of the `FPSW`.

(14) The RX Accumulator register is restored.

(15) The `QK_EXT_SAVE()` macro specifies the process of saving the extended context, which in the case of the RX processor is the FPSW register.

(16) The Renesas built-in function `get_fpsw()` resolves to a single instruction "`MVFC Rn`".

(17) The `QK_EXT_RESTORE()` macro specifies the process of restoring the extended context, which in the case of the RX processor is the FPSW register.

(18) The Renesas built-in function `set_fpsw()` resolves to a single instruction "`MVTC Rn`".

5.4 The Board Support Package for the QK Port

The BSP for the QK version is located in the file `bsp.c` in the directory `qpc\examples\rx\qk\renesas\dpp-qk-yrdkrx62n\`. The BSP for QK is mostly similar to the "Vanilla" kernel discussed before. This section describes only the differences between the two kernels.

5.4.1 Interrupt Service Routines (ISRs) for the preemptive QK kernel

As mentioned earlier, the Renesas compiler enables the user to write ISRs as interrupt functions in C/C++. These compiler-generated ISRs are perfectly adequate for the preemptive kernel

Listing 12: Tick timer ISR for QK in intprg.c

```
(1) #include "vect.h"
    #pragma section IntPRG
    . . .
(2) void Excep_CMTU0_CMT0(void) {
(3)     QK_ISR_ENTRY();          /* inform the QK kernel about entering the ISR */

    #ifdef Q_SPY
        l_tickTime_ += l_tickPeriod_;         /* account for the clock rollover */
    #endif

        QF_TICK(&l_tickISR);                   /* process all armed time events */

(4)     QK_ISR_EXIT();             /* inform the QK kernel about exiting the ISR */
    }
```

(1) The header file `vect.h` provides declarations of all possible interrupt functions.

(2) The ISR must be one of the `#pragma interrupt` functions declared in the `vect.h` header file.

(3) The macro `QK_ISR_ENTRY()` informs the QK kernel that the ISR has been entered. It **must** be called before invoking any QP services in the ISR body.

(4) The macro `QK_ISR_EXIT()` informs the QK kernel that the ISR is about to be exited. It **must** be called at the very end of the ISR body.

NOTE: The macros `QK_ISR_ENTRY()` and `QK_ISR_EXIT()` save and restore the RX Accumlator register, respectively.

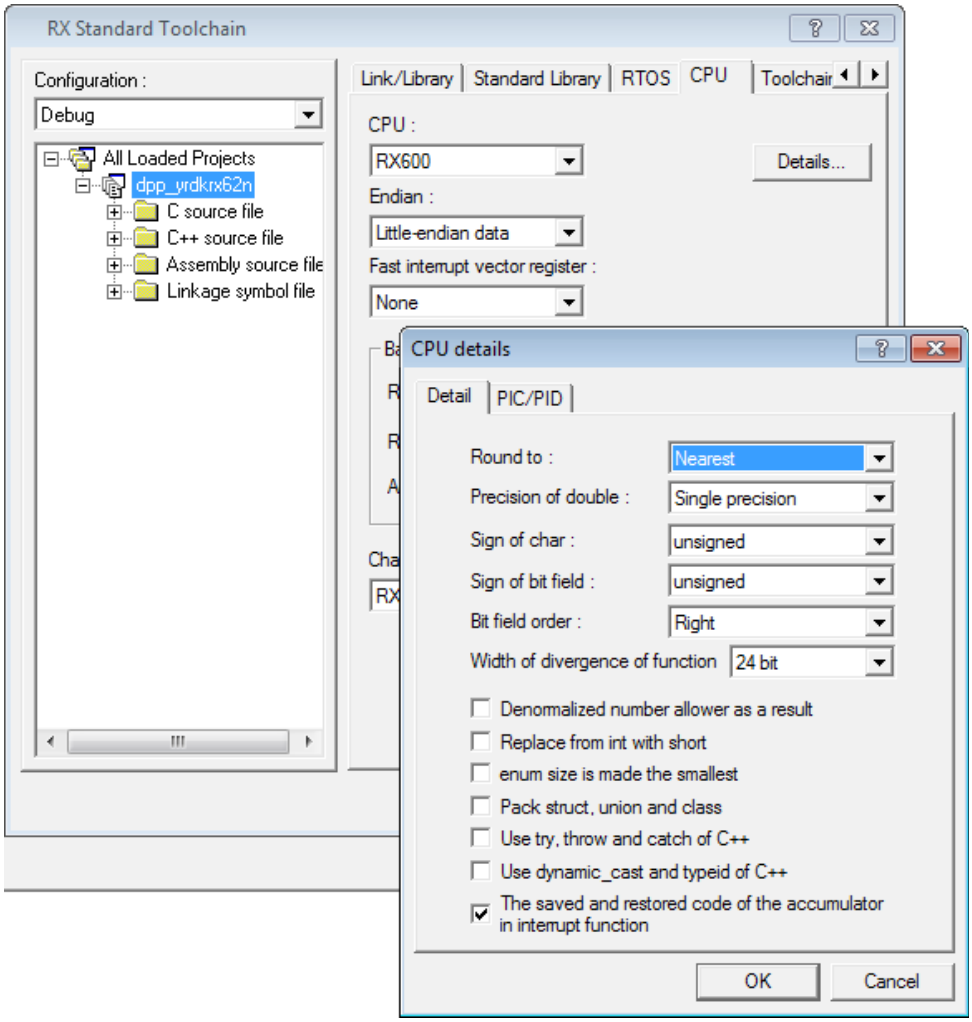
Copyright © Quantum Leaps, LLC. All Rights Reserved.

5.4.2 *Saving and Restoring the Accumulator in the ISRs*

The interrupt functions generated by the RX compiler with `#pragma vector` do **not** preserve the RX accumulator register, unless the compiler option `-save_acc` is activated. This can be achieved in the HEW IDE by selecting the menu **Build | RX Standard Toolchain**, the **CPU** tab, **Details...** button (see Figure 11).

NOTE: When using the preemptive QK kernel, you always need to activate the `-save_acc` compiler option, even if your ISRs do not perform any MAC or floating-point instructions. This is because, in QK interrupts can lead to a task preemptions, and the task-level code could be using these instructions.

Figure 11: Activating the option -save_acc in HEW



Copyright © Quantum Leaps, LLC. All Rights Reserved.

5.4.3 Idle loop customization in the QK port

As described in Chapter 10 of [PSiCC2], the QK idle loop executes only when there are no events to process. The QK allows you to customize the idle loop processing by means of the callback `QK_onIdle()`, which is invoked by every pass through the QK idle loop. You can define the platform-specific callback function `QK_onIdle()` to save CPU power, or perform any other "idle" processing.

NOTE: The idle callback `QK_onIdle()` is invoked with interrupts unlocked (which is in contrast to `QF_onIdle()` that is invoked with interrupts locked, see Section 6).

Listing 13: Idle callback QK_onIdle() for the QK kernel in bsp.c

```
(1) void QK::onIdle(void) {

(2)     QF_INT_DISABLE();
(3)     BSP_LED_IDLE_ON();      // toggle the User LED on and then off, see NOTE02
(4)     BSP_LED_IDLE_OFF();
(5)     QF_INT_ENABLE();

(6) #ifdef Q_SPY
        . . .
(7) #elif defined NDEBUG
(8)     _wait();
    #endif
    }
```

(1) The idle callback for the QK kernel is always entered with interrupts enabled.

(2) Interrupts are disabled to toggle the idle LED.

(3-4) The idle LED (LED12 of the YRDKRX62N board) is toggled on and off each time through the idle callback. Thus, a brighter LED indicates more idle time. A heavily loaded CPU will spend little or no time in the idle loop and thus the idle LED would be dim. Note that in a release build, the CPU will sleep most of the time, and the LED will be toggled only at the rate of the interrupts that wake up the CPU. In a debug build variant, the CPU never sleeps, and the LED's intensity gives a good indication of the idle task activity.

(5) Interrupts are re-enabled after toggling the idle LED.

(6) The software-tracing code is conditionally compiled into the build when enabled at compile time. When using QS, the trace data is sent out during idle loop processing. If the UART's Transmit Data Register (TDR) is empty, this code calls a QSPY API to see if there is another byte of data to send out the serial port.

(7) In release mode (no debugger) the CPU can be put to low-power sleep.

(8) The intrinsic function `wait()` provided by the Renesas compiler simply inserts a `WAIT` instruction to the RX CPU. (More advanced power management could be performed here; this port only sleeps the CPU, and does not put any other peripheral in low-power mode).

Copyright © Quantum Leaps, LLC. All Rights Reserved.

6 QS Software Tracing Instrumentation

Quantum Spy (QS) is a software tracing facility built into all QP components and also available to your application code. QS allows you to gain unprecedented visibility into your application by selectively logging almost all interesting events occurring within state machines, the framework, the kernel, and your application code. QS software tracing is minimally intrusive, offers precise time-stamping, sophisticated runtime filtering of events, and good data compression (please refer to “QSP Reference Manual” section in the “QP/C Reference Manual” and also to Chapter 11 in [PSiCC2]).

This QDK demonstrates how to use the QS to generate real-time trace of a running QP application. Normally, the QS instrumentation is inactive and does not add any overhead to your application, but you can turn the instrumentation on by defining the Q_SPY macro and recompiling the code. In this QDK, this means selecting and building the "Spy" variant of the code (as opposed to the Debug or Release variant).

QS can be configured to send the real-time data out of the serial port of the target device. On the YRDKRX62N evaluation board, QS uses SCI2 in asynchronous (UART) mode to send the trace data out of DB9 connector J5 "COMM PORT". The YRDKRX62N kit does not come with a serial cable, you will need to provide this. In all likelihood, you will need to use a serial-to-USB cable/adapter as nowadays it is difficult to find a computer with a standard DB9 serial port.

NOTE: This port uses the serial port (configured for 115,200 baud by default) as the output port for sending QSPY trace data.

6.1 QS initialization in QS_onStartup()

The routine `QS_onStartup()` is invoked during initialization to initialize and configure the QSPY tracing buffer, timing variables, and trace data transmit UART.

Listing 14: QS initialization in bsp.c

```
(1) uint8_t QS_onStartup(void const *arg) {
(2)     static uint8_t qsBuf[6*256];                 /* buffer for Quantum Spy */
(3)     QS_initBuf(qsBuf, sizeof(qsBuf));

                        /* Configure & enable the serial interface used by QS   */
(4)     . . .

        /* Initialize QS timing variables */
(5)     QS_tickPeriod_ = TICK_COUNT_VAL;
(6)     QS_tickTime_   = 0;                    /* count up timer starts at zero */
(7)     QS_FILTER_ON(QS_ALL_RECORDS);
(8)     QS_FILTER_OFF(QS_QF_ACTIVE_ADD);
        . . .
        QS_FILTER_OFF(QS_QF_ISR_EXIT);

(9)     return (uint8_t)1;                                  /* return success */
    }
```

(1) The `QS_onStartup()` routine is called when the BSP is initialized through the macro QS_INIT().

(2) Trace data to be sent out the serial port is enqueued in this buffer. Adjust the size of this buffer based on your application's tracing needs. The bursts of volume of data being traced / transmitted, and the serial baud rate will impact the size required.

(3) `QS_initBuf()` is called to initialize the output buffer.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

(4) This omitted piece of code performs the hardware-specific initialization of the UART used by QSPY. QSPY uses the YRDKRX62N's single serial comm port, which is connected to Serial Communications Interface #2 (SCI2) on the RX62N. Chip-specific details of the UART initialization are not shown here, but they are heavily commented in the source code and reference sections in the chip's manual.

(5-6) `QS_tickPeriod_` is initialized and never changes. This is the value that timer peripheral CMT0 (used for the OS tick timer) counts up to before matching & resetting to zero before counting up again. `QS_tickTime_` increments by an amount of `QS_tickPeriod_` each time a kernel tick timer interrupt is handled. The description of `QS_onGetTime()` later describes use of these values.

(8) Here we enable all QS trace records, and then selectively disable some of them. Deciding which data to trace is a subject covered in depth in Chapter 11 of [PSiCC2] and is not specific to this port.

(9) We return 1 to indicate initialization was successful.

6.2 QS Trace Output in QF_onIdle()/QK_onIdle()

To be minimally intrusive, the actual output of the QS trace data happens when the system has nothing else to do, that is, during the idle processing. The following listing shows the code placed either in the `QF_onIdle()` callback ("Vanilla" port), or `QK_onIdle()` callback (in the QK port):

Listing 15: QS trace output in bsp.c

```
    void QF_onIdle(void) {
        . . .
(1) #ifdef Q_SPY

                    /* RX62N has no TX FIFO, just a Transmit Data Register (TDR) */
(2)     if (SMCI2.SSR.BIT.TDRE) {                        /* Is SMC's TDR Empty? */
(3)         uint16_t b;

(3)         QF_INT_DISABLE();
(4)         b = QS_getByte();
(5)         QF_INT_ENABLE();

(6)         if (b != QS_EOD) {
(7)             SMCI2.TDR = (uint8_t)b;               /* send byte to UART TDR */
            }
        }
    . . .
```

(1) The code is only compiled into the Q_SPY build configuration.

(2) The UART's TDRE status flag is checked to see if the Transmit Data Register is Empty.

(3) Interrupts are disabled to protect the QS output trace buffer while it is being accessed from non-interrupt code. Otherwise, this idle loop processing could be interrupted by an ISR which could try to access the trace buffer and corrupt it (shared resource problem).

(4) The temporary variable for storing the return from the `QS_getByte()` API call must be 16-bit wide.

(4) The `QS_getByte()` returns the next byte of trace data or `QS_EOD` if no data is available.

(5) Once access to the trace buffer is complete, interrupts can safely be re-enabled.

(6) If the returned data is not `QS_EOD` (end of data), the data needs to be sent out

(7) A byte of trace data is written into the Transmit Data Register and sent out the UART.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

6.3 QS Time Stamp Callback QS_onGetTime()

The platform-specific QS port must provide function `QS_onGetTime()` (Listing 9(12)) that returns the current time stamp in 32-bit resolution. To provide such a fine-granularity time stamp, the RX port uses the CMT0 facility, which is the same timer already used for generation of the system clock-tick interrupt.

NOTE: The `QS_onGetTime()` callback is always called with interrupts disabled.

Listing 16: QS initialization in bsp.c

```
(1) QSTimeCtr QS_onGetTime(void) {            /* invoked with interrupts locked */
(2)     if (IR(CMT0, CMI0)) {                          /* is tick ISR pending? */
(3)         return QS_tickTime_ + CMT0.CMCNT + QS_tickPeriod_;
        }
        else {                                          /* no rollover occured */
(4)         return QS_tickTime_ + CMT0.CMCNT;
        }
    }
```

(1) `QS_onGetTime()` is a very important function. It provides a high-resolution time-stamp for outgoing trace records. It is invoked at the time a trace record is appended to the trace record by the application (as opposed to when the record is transmitted). This routine is called with interrupts disabled.

(3) We check to see if the tick timer interrupt on timer peripheral CMT0 is already pending. (it could be pending, because interrupts are disabled inside the `QS_onGetTime()` callback).

(4) If timer peripheral CMT0 has a pending interrupt, this indicates that the free-running counter CMCNT has already counted up to its target match value and has been re-loaded with 0 again. Thus, in order to get an accurate timestamp, the overflow must be accounted for by adding an extra number of ticks (stored in `QS_tickPeriod_`) in addition to the step described in item

NOTE: The CMT0 interrupt (the system clock tick) could be pending at this time, because interrupts are disabled. Once interrupts are re-enabled, the timer interrupt will fire, `QS_timeTick_` will be incremented by `QS_tickPeriod_` in the tick ISR as we have seen, and the timer interrupt pending flag will be cleared.)

(5) If the timer interrupt is not pending, the current timestamp is the sum of the long-term value in `QS_timeTick_`, which is updated in fairly large increments, and the instantaneous value in the free running timer counter register CMCNT in peripheral CMT0. The counter is an up-counter, so this is a simple addition.

6.4 Running the QSpy host application

The QSPY host application receives the QS trace data from the target, parses it and displays on the host workstation (currently Windows, Linux, or Mac OS X). For the configuration options chosen in this port, you invoke the QSPY host application as follows (please refer to “QSP Reference Manual” section in the “QP/C Reference Manual” and also to Chapter 11 in [PsiCC2]):

```
qspy -cCOMx
```

NOTE: COMx will be something like COM1, COM2, etc. depending on your machine's serial port availability and configuration.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

7 Related Documents and References

Document	Location
[PSiCC2] "Practical UML Statecharts in C/C++, Second Edition", Miro Samek, Newnes, 2008	Available from most online book retailers, such as Amazon.com. See also: http://www.state-machine.com/psicc2.htm
[QP/C 08] "QP/C Reference Manual", Quantum Leaps, LLC, 2010	http://www.state-machine.com/doxygen/qpc/
[QP/C++ 08] "QP/C++ Reference Manual", Quantum Leaps, LLC, 2010	http://www.state-machine.com/doxygen/qpcpp/
[QL AN-Directory 07] "Application Note: QP Directory Structure", Quantum Leaps, LLC, 2007	http://www.state-machine.com/doc/AN_QP_Directory_Structure.pdf
[QL AN-DPP 08] "Application Note: Dining Philosopher Problem Application", Quantum Leaps, LLC, 2008	http://www.state-machine.com/doc/AN_DPP.pdf
RX Family C/C++ Compiler, Assembler, Optimizing Linkage Editor Compiler Package User's Manual	Document EUM_CCRX_v100_rej10j2062.pdf available in the HEW distribution for Renesas RX.
GNU-RX Compiler Collection, KPIT Cummings	Documentation in HTML available with the GNU-RX (KPIT) toolset at Hew/Tools/KPIT/GNURX-ELF/v12.01/Doc/gcc.html. See also http://www.kpitgnutools.com
Renesas Demonstration Kit (YRDKRX62N) for RX62N User's Manual: Hardware [REU10B0009-0100]	Available from the YRDKRX62N kit CD-ROM & downloaded from Renesas website as well
RX621 Group: Datasheet [REJ03B0281-0050]	Available from the YRDKRX62N kit CD-ROM & downloaded from Renesas website as well
RX62N Group, RX621 Group: User's Manual: Hardware [R01UH0033EJ0111]	Available from the YRDKRX62N kit CD-ROM & downloaded from Renesas website as well
RX62N Group, RX621 Group: User's Manual: Software [REJ09B0435-0100]	Available from the YRDKRX62N kit CD-ROM & downloaded from Renesas website as well

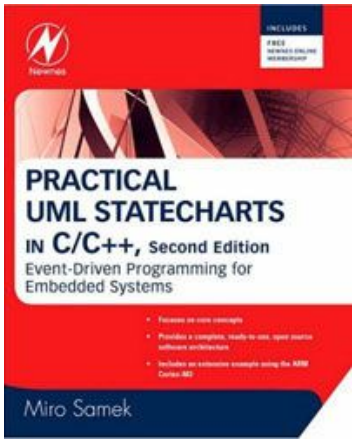
Copyright © Quantum Leaps, LLC. All Rights Reserved.

8 Contact Information

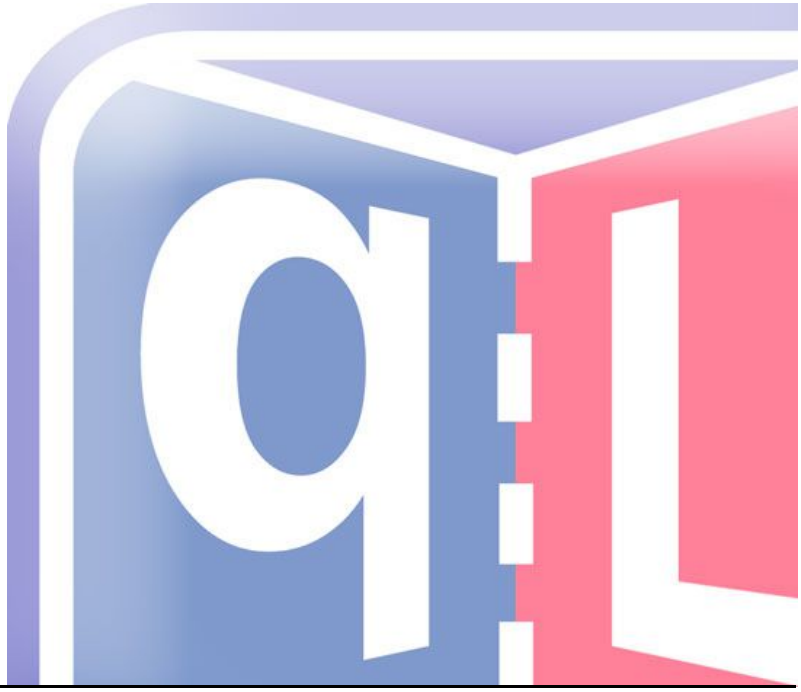
Quantum Leaps, LLC
103 Cobble Ridge Drive
Chapel Hill, NC 27516
USA

+1 866 450 LEAP (toll free, USA only)
+1 919 869-2998 (FAX)

e-mail: info@quantum-leaps.com
WEB : http://www.quantum-leaps.com
http://www.state-machine.com



"*Practical UML Statecharts in C/C++, Second Edition: Event Driven Programming for Embedded Systems*", by Miro Samek, Newnes, 2008



Copyright © Quantum Leaps, LLC. All Rights Reserved.